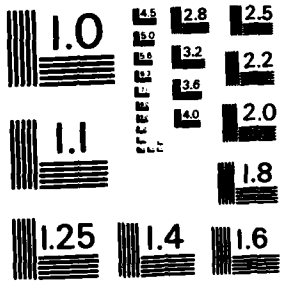
AD-A127 310 THREE APPROACHES TO TESTING THEORY(U) MARYLAND UNIV 1/1
COLLEGE PARK DEPT OF COMPUTER SCIENCE D HAMLET DEC 82
TR-82/15 AFOSR-TR-83-0304 F49620-80-C-0001

UNCLASSIFIED F/G 14/2 NL



THREE APPROACHES TO TESTING THEORY

Dick Hamlet

Technical Report 82/15
December, 1982

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF MELBOURNE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR- 83-0304	2. GOVT ACCESSION NO. AD-A127 3(1)	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) THREE APPROACHES TO TESTING THEORY		5. TYPE OF REPORT & PERIOD COVERED TECHNICAL
		6. PERFORMING ORG. REPORT NUMBER TR #82/15
7. AUTHOR(s) Dick Hamlet		8. CONTRACT OR GRANT NUMBER(s) F49620-80-C-000/
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer Science University of Maryland College Park MD 20742		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS PE61102F; 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Mathematical & Information Sciences Directorate Air Force Office of Scientific Research Bolling AFB DC 20332		12. REPORT DATE DEC 82
		13. NUMBER OF PAGES 12
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)
Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
The relationship between program and specification is the core of programming theory. Since both program and specification are finite representations of infinite objects (the input-output behavior implemented and desired), it is not surprising that the relationship is not an effective one. Testing theory is concerned with the implications of limited program executions, with the ultimate goal of deciding a program's correctness. The difficulty at the heart of testing for correctness is that the class of programs computing each function is not algorithmically recognizable. With a functional (CONTINUED)

DD FORM 1 JAN 73 1473

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ITEM #20, CONTINUED: specification, no test can distinguish correct programs from all others. However, this problem can be circumvented: (1) by requiring human assistance in selecting tests, (2) by strengthening specifications, or (3) by treating tests as statistical samples. Each approach has also contributed useful insight to practical testing.

The primary goal of testing is explanation. Many testing methods seem to work in practice, but there is no theoretical reason for their good performance. Experiments to validate testing methods are of doubtful validity because it is difficult to distinguish the contributions of people and the programming/testing process from those of the method. Too often success in practice can be traced to an accident of human intuition, in which the testing scheme is incidental.

THREE APPROACHES TO TESTING THEORY

Dick Hamlet

Technical Report 82/15
December, 1982

Department of Computer Science
University of Melbourne+
Parkville, Victoria 3052
Australia

Abstract

The relationship between program and specification is the core of programming theory. Since both program and specification are finite representations of infinite objects (the input-output behavior implemented and desired), it is not surprising that the relationship is not an effective one. Testing theory is concerned with the implications of limited program executions, with the ultimate goal of deciding a program's correctness. The difficulty at the heart of testing for correctness is that the class of programs computing each function is not algorithmically recognizable. With a functional specification, no test can distinguish correct programs from all others. However, this problem can be circumvented: (1) by requiring human assistance in selecting tests; (2) by strengthening specifications; or (3) by treating tests as statistical samples. Each approach has also contributed useful insight to practical testing.

The primary goal of testing theory is explanation. Many testing methods seem to work in practice, but there is no theoretical reason for their good performance. Experiments to validate testing methods are of doubtful validity because it is difficult to distinguish the contributions of people and the programming/testing process from those of the method. Too often success in practice can be traced to an accident of human intuition, in which the testing scheme is incidental.

+On leave from Department of Computer Science, University of Maryland, College Park 20742 U S A. This work was partially supported by the United States Air Force Office of Scientific Research under grant F49620-80-C-[illegible].

1. Introduction

As a formal discipline, testing theory dates only from the mid-1970s. Even program proving is older, beginning with the informal assertions advocated by von Neumann and Turing, which were formalized in Floyd's 1967 paper. Programmers have always done testing, and have perhaps believed that it showed their programs worked, but formal analysis was late in coming. Indeed, the idea of correctness is logically prior to any testing theory, since it is required to define what it means for a program to work. Perhaps a reason for disinterest in testing theory is the widespread belief expressed in Dikjstra's aphorism, "testing can only show the presence of errors, never their absence." When a test fails, no theoretical principles are needed to see the implications; when tests succeed it seems equally clear that little has been proved.

But the intuition persists that something has been proved by testing. Programs do contain faults, and a method of finding some of them will always be useful. Furthermore, a fault-finding method can be evaluated for how well it pinpoints difficulties, and how often it surprises the confident programmer who knows it will be used. The other side of the testing coin is the ability of a test to increase confidence in the correctness of the program that passes it. There is an unsolvable problem at the core of confidence testing, but three distinct lines of investigation appear promising:

(I) Tests are used only as part of a creative, nonmechanical proof method. Human beings do the work, but their task is made more natural by enlisting the additional information of a test success.

(II) Program specifications can be strengthened so that correctness (relative to the new, more restrictive specification) can be established algorithmically. Conventional testing tools can be adapted to monitor conformity to strengthened specifications.

(III) A statistical theory can establish probable correctness, within confidence limits reflecting the number of tests conducted. Testing tools and methods may be analyzed and compared by this technique, and it has direct practical promise.

The process of programming and of program testing are linked by the programmer's prior knowledge that code will be tested, and by test results during debugging. The intimate involvement of clever, dedicated people with program and test makes it difficult to evaluate test methods objectively. Experiments conducted to establish or compare the efficacy of methods can be entirely misleading, because they fail to account for human involvement in creating and debugging the program. For example, so-called "exercise" testing, in which test coverage of program fragments is monitored, is subject to the "nose-rubbing effect."

Consider the simplest coverage criterion, that every statement of a program must be used in some test. The conventional explanation for why statement-coverage testing works is that further tests required to cover unused statements make it more difficult for faults to go unnoticed. But this explanation ignores the human analysis that goes into finding additional tests. The unexecuted statements are examined to see what they do, and this analysis may well uncover some fault in them. Since they have not been executed, the credit for finding this fault certainly goes to the person, not the test method. Once their purpose is understood, other parts of the

software must be analyzed to see why they have not been used, and this requires examination of paths and predicates that might lead there. In the process the tester looking for trouble is liable to find it, but not necessarily trouble related to statements not executed. (Perhaps for the omitted case that left statements unexecuted, the software works perfectly.) "Nose rubbing" is a good name for the process of directed human error finding: a technical expert, looking for deficiences, is forced to examine particular parts of the code, and there (or nearby) faults are found. Code being what it is, close study almost always finds bugs.

To validate a testing method instead of the human being using it would require separating faults found "nearby" from those that directly cause test deficiencies; no such study has been conducted. It might also be instructive to conduct a reverse study, in which pure nose rubbing was evaluated: an arbitrary pointer into the code would be printed with an obscure error message, and people asked to fix the fault.

2. Proving-based Testing Theory

In the proving view, the goal of testing theory is to find test ideas equivalent to correctness, ideas to be established for a program in noneffective ways, so that the success of a "good" test implies correctness. The complete, reliable, and valid notions of [1] are of just this kind. A cynic might say that the difficult proof of correctness has been traded in for an equally difficult proof of test "goodness," but this need not be so. Success of the test is presumed, which may remove technical difficulties from the proof [3]. The basis for proving-based testing theory is the idea of "treated the same."

Because programs and specifications are finite, there may be a division of the input space into finitely many classes that are (or should be) "treated the same." In that case, it is sufficient to select one element from each class, and test success on these elements implies that the program is correct. The difficulty lies in defining "treated the same" so that the formal version of this argument is valid. Intersecting input classes with diverse origins seems intuitively to approach "treated the same," but the finest partitions can be inadequate, until the classes contain but a single element.

As an example of the intersection of input partitions, consider the "triangle problem" of classifying triples of integers (A, B, C) representing the sides into the textbook types such as "obtuse scalene." The outputs specified form a finite set, and are thus a natural source of one input division. Suppose the program for this problem has no loops, so that its path equivalence classes are of finite index as well. Intersecting these partitions yields input classes in which a certain classification is obtained by traversing a fixed path. Is it possible to select a successful test yet not have a correct program? Evidently so, since the path predicates can be wrong, but accidently correct for selected tests. The intuitive satisfaction we feel with a fine input division is perhaps a confusion between the fault-finding and confidence roles of testing: should there be an error, attention is directed to a narrow case.

Programs like the triangle-classification one, in which there is a simple description of the required output, and the program contains no loops that cannot be unwound, are not good examples on which to try testing methods, because they yield perfectly to symbolic execution [4]. It is only necessary to prove that the path predicates in disjunction exhaust the input space, and that each corresponds to the triangle type that is identified on that path.

For example, a path predicate

$$A + B > C \;/\backslash\; A < B < C$$

must be on a path where "obtuse scalene" is printed. These theorems are often within the abilities of mechanical theorem provers.

The role of "treated the same" input classes is different when the classes appear in program and specification. If specifications are not prescriptive of how results are to be obtained, those inputs that should be treated the same need not be in fact. For example, in computing the absolute value function, it is common to specify the behavior differently for positive and negative inputs. But the programmer who writes

```
real procedure absval(x); real x; value x;
  absval := sqt(x^2)
```

is not observing the positive-negative distinction, and those classes have no significance for this program. Again there has been a confusion with the process of programming, and the fault-finding role of testing: a programmer working from specifications is most likely to go wrong on their boundaries, often by assuming an unwarranted "continuity." In fault-finding tests, those boundaries are therefore significant, but they may be meaningless in confidence testing.

Input partitions defined by program and those defined by specification can be very similar, or very different, depending on how prescriptive the specification is. When the partitions are very different (say because the specification is declarative rather than operational), the boundaries in each are reflected into the other in surprising ways, and this may be an advantage in fault finding. When the partitions are similar, the boundaries seem to have more significance in confidence testing. If the specification is prescriptive, its boundaries are required to coincide with the program's, a requirement that can be checked to aid in fault finding. None of these arguments seem to bear on the question of whether specifications should be similar to programs or not. The major practical points are that operational specifications seem easier to write; but, it is also easier to make the same mistake when translating intuition into an operational specification and program, a situation no testing or proving method can detect. For testing theory, it seems best to begin with the simplest case and take care with definitions.

Testing based on input partitions for identical output does lead to correctness, as the following trivial theorem shows. Define the same-output equivalence relation for a program as

$$P^{=} = \{(x, y) \mid \text{inputs } x \text{ and } y \text{ lead to the same output}\}.$$

Define the same-output equivalence relation for a specification as

$$S^{=} = \{(x, y) \mid \text{outputs on inputs } x \text{ and } y \text{ should be the same}\}.$$

The input classes defined by the intersection of $P^{=}$ and $S^{=}$ are those that should have the same output, and do in fact. Initially, let the members of these intersection classes be called "treated the same."

Theorem: A test using one arbitrary element from each treated-the-same input class is successful iff the program is correct.

Proof. (Correctness as a consequence of test success.) Let x be any input,

which lies in treated-the-same class Y. Some member of Y has yielded the correct result, by definition of a successful test. The point x is supposed to give this result, and does in fact, by definition of Y. Hence the program is correct for input x. (The other implication is trivial.)

Corollary: $P^{=}$ and $S^{=}$ are the same iff the program is correct.

Proof. The identity of $P^{=}$ and $S^{=}$ and the success of an arbitrary test from each class, are the same in the Theorem.

The evident deficiency in this result is that the treated-the-same classes are not often of finite index; it is also important that there may be no way to obtain a representative of each class. In problems like triangle classification where the index is finite, choosing a point from a specification class (like "equilateral") may be easy, and a successful test execution shows that the intersection with the "equilateral" program class is not empty. But it does not prove correctness to proceed in this way, because there is no effective way to select points in error classes (such as: specified "equilateral" but the program prints "right isosceles")--indeed, the Corollary states that these classes must be empty for correctness.

Thus in its simplest form, use of treated-the-same input classes is a proving technique that makes no use of testing at all: once the error classes have been shown to be empty, there is no need to try points in the others, which must coincide.

There are natural input partitions for specifications and programs broader than $S^{=}$ and $P^{=}$. If these are more likely to be of finite index, or have easier to find representatives, they are candidates for a testing method. For example, suppose a first-order logic specification is of the form

$$I_1(x) \ , \ O_1(x,y)$$
$$I_2^1(x) \ , \ O_2^1(x,y)$$
$$\cdots$$
$$I_n(x) \ , \ O_n(x,y)$$

where the I_i are disjoint input assertions, and the corresponding O_i are output assertions for those inputs. Let I be the disjunction of all the I_i, and

$$R_i(x) = I_i(x) \ /\backslash \ \exists y \ O_i(x, y)$$

for each I_i and corresponding O_i. Then the relation

$$\{(u, v) \mid R_i(u) \ /\backslash \ R_i(v) \ \text{ for some } \ i \}$$

has input classes that intuitively follow the assertions of the specification. (The inputs not in any such class are the ones where the specification fails to constrain the result at all, because $\neg I$ holds; and, those for which the specification asks the impossible, because there are no outputs as required.) Similarly, let

$$Q_i(x) = \text{on input } x \text{ the program gets output } y, \text{ and } O_i(x, y) \ ,$$

so

$$\{(u, v) \mid Q_i(u) \ /\backslash \ Q_i(v) \ \text{ for some } \ i \}$$

has input classes for which the program's result satisfies one of the output assertions. (These program classes are not necessarily disjoint.)

Intersecting specification-defined and program-defined classes, we obtain classes that are "treated the same" in a wider sense than that used above. The same results hold (with the Corollary now stating that the intersection partition must be the same as the specification one), and there is the same difficulty in test selection. The "error classes" are now those for which some input assertion holds, but the program output fails to satisfy the output assertion, and these are not easy to identify. If they can be shown to be empty, the proof of correctness is complete without recourse to tests.

A testing method might approximate a "treated the same" input division by identifying as many of the partitions as possible, and placing a test in each. Each success proves that the entire partition is treated correctly; the approximation is that without knowledge of the extent of the untried classes, one does not know much about the correctness of the program. The situation is perhaps better than one in which undisciplined testing is done, because the existence of the assertions does implicitly describe the untried classes. On the whole, testing in this way tends to support Dijkstra's position, however.

The property that "treated the same" partitions lack might be called decidable coverage. A partition has this property if there is an algorithm to decide if an arbitrary set of inputs includes a representative from each of its classes. Partitions for which the Theorem above holds cannot have this property, or they would solve the confidence testing problem. However, program structure is a source of partitions that do have decidable coverage. For example, the program relation

{(x, y) | x and y cause the same branch to be taken}

defines an input partition whose covering representatives constitute "branch coverage" of the program. If we stipulate that the program halts for each of a given set of inputs, then the branch-coverage partition has decidable coverage, by trial. However, it is not in general a refinement of any partition for which the "treated the same" theorem holds. For example, the inputs taking a certain branch may sometimes lead to results satisfying an output predicate, and sometimes not.

Both specification and program may have partitions which fail to refine "treated the same" classes. When program and specification partitions are nearly the same, the classes formed by their intersection can be useful as a basis for proving correctness. For example, in the partition analysis method [15] the specification is procedural, and the partitions are based on path equivalence. Symbolic-execution techniques can then be used to demonstrate that within each class, inputs are treated correctly. (But they are not of course "treated the same.") In this method tests seem to play no role except as a check on the proofs and on the accuracy of subsequent transcription, compilation, etc. On the other hand, when specification and program are in different forms, intersecting partitions creates classes that are precise for fault finding, but difficult to relate to correctness. In a system with axiomatic specifications and conventional programs [16], partitions based on statement coverage and simple expression mutation have decidable coverage, but were found in practice to be quite difficult to cover, giving the nose-rubbing effect maximum play.

This analysis shows that the idea of "treated the same," while useful for fault finding because of the way programs are written from specifications, is not so useful for a confidence testing theory. Correctness turns on empty error partitions that are difficult to identify, and their identification is not aided by successful tests on identifiable partitions. When partition-class representatives can be identified, the sense of "treated the same" is not strong enough to yield correctness, unless coupled with additional

prescriptive conditions that make program and specification nearly the same. Apparently, methods utilizing partitions work because of subtle links between specification and program, or because of non-methodical inspiration in choosing the classes.

3. Testing to Strengthened Specifications

Where proving-based testing theory begins and ends with noneffective collections of tests, where the hard work is human analysis and the program test executions are almost incidental, an alternate view is exactly the reverse. In this view, which arises from testing tools developed over the past 20 years, the essential element of testing is its machine-intensive, automatic character. In both the fault-finding and confidence-testing situations, good tests are those an algorithm can generate (or at least recognize, if they must be generated by hand). This necessarily makes "goodness" different from correctness. For example, the path-coverage of [2], if defined as "good," can be easily monitored, but has no implication for correctness.

The emphasis on tools allows distinctions to be made among fault-finding methods. Not all test failures are the same. A testing tool that only reports incorrect output is obviously inferior to one that makes a successful attempt to localize the fault. (The situation has many parallels with syntax error messages generated by compilers. For example, "MISSING 'END'" requires study of a whole program, while "ILLEGAL SYMBOL" with a pointer to the scanned position, is easy to repair.) Even when a test succeeds, it is still possible to fault it as inadequate, that is, to complain not about the program but the test. All structural testing tools (of which the path analyzers [4] are the most common) do this. Failings in the test data are thereby linked to parts of the program, which may aid in discovering better data or in changing those parts. (But see the discussion of the "nose-rubbing effect" in Section 1.) Each structural criterion may be viewed as an attempt to sidestep an unsolvable problem that makes a successful test worthless, and in this sense structural coverage approximates correctness. For example, all-statements-executed coverage closes off the loophole of unreachable code. It cannot be decided if code is unreachable in general, and so long as code remains unused it could contain faults. By creating an artificial test failure with a pointer to unused code, a testing scheme cuts through this problem. (Similarly the idea of "mutation" [5] closes the loophole of expressions that are incorrect, but by chance correct for given tests [6].)

The unsolvable problem to which confidence testing is reduced is that of program equivalence. Because the collection of programs computing one function is not recognizable by algorithm, neither is it possible to recognize a test as guaranteeing correctness. The problems of an algorithmic testing theory thus arise from (1) flexible programming languages in which there exist an unrecognizable set of programs with the same functional behavior, and (2) the choice of functional specifications.

In many cases software engineering problems can be solved by imposing useful structure on a chaotic situation. For example, the problems of understanding "spaghetti code" can be overcome by arbitrarily banning the GOTO; the unsolvability of detecting infeasable control paths can be circumvented by requiring tests to display full path coverage, etc. It might therefore seem reasonable to look for a way of controlling the set of programs with the same functional behavior. Unfortunately, program equivalence will be undecidable in any language able to compute all primitive recursive functions, for which there is an interpreter and a way to mechanically compose programs

[9]. It is unlikely that any of these properties can be eliminated from a usable programming language.

The program-equivalence problem has an algorithmic solution when "equivalence" is defined in a more restricted way than the functional. For example, in a programming language capable of computing any subset of the total recursive functions, the set of programs with the same computation function is recursive [10]. This suggests that by strengthening specifications, it may be possible to establish by test that a program meets them. Furthermore, algorithms that recognize computational equivalence do so on the basis of a finite specification, which is itself restricted to the test domain. This solves the problem of the specification oracle. Two examples will show the character of strengthened specifications.

Consider assembly-language programs in the idealized model popularized by Minsky as "register machines" [11]. (Any model of computation will serve for the proof, but the details are easier if the "steps" of computations are strictly finite, as in Turing machines, but not in a language allowing arithmetic on arbitrary-size operands as a "step.") Any assembly-language program can be reconstructed from a finite sample of its computations (up to register names, and the relative placement of basic blocks of instructions), if the program does not contain unreachable instructions. The necessary inputs for this computation set can be imagined as obtained by executing the program on an arbitrary input, and recording which instructions are used and in which sequence. Because each instruction has a unique effect displayed in the computation, this fixes part of the program. If there are no open conditional branches, the single computation is sufficient. If a branch is open, then an input must be found to take that branch, and thus fix the code branched to. This process continues until the finite number of branches of the program have all been closed, and hence the whole fixed. There is no algorithm for generating the set of computations because the problem of identifying unreachable instructions is unsolvable. A branch might remain forever unclosed, and no systematic exploration of the input space will help to close it. However, presented with a purported set of computations that fixes a program, the claim can be tested efficiently by executing the program, and seeing if those computations result, and exhaust its instructions.

The finite collection of computation traces that fix any program without unreachable instructions constitutes a strengthened specification of that program. This specification involves a finite number of inputs, and provides far more information about what the program must do on those inputs than would an input-output specification. However, it fails to provide information about what the program should do on other inputs, save by yielding the program for experimentation. The character of the specification is that the program must behave precisely on a finite set; elsewhere, its behavior is determined, but not explicitly described. If two programs meet the specification they necessarily have identical computations everywhere, because up to inessential register names and instruction rearrangement, they are the same program.

A second kind of strengthened specification has the same peculiar character. Take the specification to be a finite collection of input-output pairs, plus a restriction on the form of the program that is to realize this behavior. Let the program restriction guarantee that all programs halt for all inputs. (For example, loops might be required to be a priori bounded, as in LOOP [9].) Also let a program exist for each finite function (perhaps using table lookup). The smallest program that realizes the given finite sample of behavior is then well defined, and may be found by trial and error. (Since all permitted programs terminate on all inputs, they can be tried in order of size; the finite-function program will stop the search eventually.)

In this simplified form, it may seem unlikely that such specifications would lead to the desired program, but more elaborate program restrictions can produce surprising results, both in the speed with which the desired program can be found, and its entire behavior. Summers's automatic programming system [12] is particularly impressive in this regard, for example. It constructs programs that use simple recursion in a natural way, so that if there is a simple recursive solution, a remarkably small volume of data will cause it to be generated.

Specifications by example and program restriction have the same character as those by determining computation: for the test points given, the behavior is precisely prescribed; but, for other points it is not even indicated. For example, in Summers's system, the program induced is the "most natural" (in a certain technical sense) generalization of the examples, but it is easy to imagine being surprised by the behavior of this program on untried input. It may be that the specification cannot be realized in the restricted program form, or that the part of the algorithm that makes the realized program unique (the "least" condition in the simple definition above) gets the wrong program. In the latter case, there is no simple way to extend the supplied data to force the generation of a different program, and of course it is an unsolvable problem to distinguish insufficient data from an impossible specification.

Conditions that select one program among many possible ones are an essential part of any automatic programming system, one more indication that the program-equivalence problem lies at the heart of testing difficulties. Arbitrary selection conditions make the relationship between testing and generating programs a little more complex than it seems. If one wants to decide if a particular program meets a strengthened specification, there is the obvious procedure of generating the program that does meet it, to compare with the given one. But *this procedure will fail to certify programs* equivalent to the generated one but not identical to it. When programs are made by people and tested, it is unlikely that the strengthened-specification form will be observed exactly, and such programs then cannot be certified by test. The point is moot in practice, since if the automatic programming algorithm is judged appropriate, there is no reason not to use it in the first place, and avoid testing altogether. That people probably could not program according to the automatic algorithm, points up how unsure we are of what such a procedure is actually doing, however.

Weyuker has proposed a method of assessing test data adequacy [13] that is proving-based, but also uses the idea of strengthened specifications. Given a program, specification (input-output only), and a set of test data, the latter is called <u>inference adequate</u> iff a program generated from the tests (say by Summers's scheme): (a) is equivalent to the given program, and (b) meets the given specification. (Her method is proving based because neither of these equivalences can be established mechanically.) The success of this method evidently depends on the ability of the inference scheme to generate programs meeting the specifications. It is not true that it would be as well to prove the given program to meet the specification, because the restricted form of the generated program may make this proof easier.

Weyuker also suggests an effective approximation to inference adequacy as follows: a program is inferred from the given data, and then tested using new data unrelated to the set used for inference. On this new data it must agree with both the original program and specification. In this process the inferred program plays no role--the result is not different from selecting two independent test sets. The peculiar form of the inferred program may make test selection easier, and if the newly selected data fails, the intermediate program may help in assigning the fault to original program or specification. However, it seems useful to add to Weyuker's method the requirement that when

the new data is added to the original, the same program be inferred. (Or, if a proving-based method is acceptable, that the two inferred programs be equivalent.)

The idea of comparing programs inferred from different sets of test data leads to the following notion of test equivalence that can be made effective in some practical cases: two test sets are equivalent if the programs inferred from them are equivalent. The effectiveness of the idea depends on the form of the inferred programs. For example, using Summers's program generator, equivalence may be decidable using techniques invented by Samet [14], which exploit the special form of the resulting programs. This idea of equivalence is weaker than the one of adequacy that Weyuker wishes to define, but it also suggests an idea for probabilistic analysis: if successive expansions of a test yield only equivalent tests, is it not more likely that the original was adequate? This notion can be used whenever test equivalence has an effective definition but test adequacy does not.

4. Random Testing Theory

The nose-rubbing effect is a likely explanation for any success that involves human beings working in conjunction with a testing tool. To otherwise explain the success of tools requires a statistical theory. Tests are samples, drawn from the space of all behaviors the software can exhibit; it should be possible to estimate the software's characteristics from these samples, and to further calculate the confidence to be placed in those estimates. There are two difficulties: (1) the space of all behaviors is large, so it may be impractical to amass a significant sample; and (2) test points must be selected in a way that guarantees "independence" if their significance is to accumulate as the number of points grows. Viewed in one way, both of these problems have straightforward solutions; from another viewpoint the necessary research work remains to be done.

A straightforward random testing theory considers distribution of tests across the input space [17]. The essential assumption is that the program has an "operational input space," from which points can be drawn at random, and that success on any point selected is independent of that on any other point. (The input space may be partitioned [18], but we have seen in Section 2 that this idea only lends a misleading refinement.) If n points are selected, and if the behavior is in error for k of them, then the software failure rate will tend to k/n as n grows. If $k = 0$, and one requires $1 - e$ confidence that the probability of no errors ever appearing is p, then

$$n = \frac{\log e}{\log p}$$

points are required [8]. For example, to attain 90% confidence that the probability of no faults is .95 requires the success of 45 test points chosen independently from the operational input space.

It is striking that the result of this simple theory is the same for all programs, and that the number of test points required for great confidence is so small. These counter-intuitive properties of the theory are consequences of the assumption of independent test selection from an operational input space, which evidently requires further analysis. In practice, the operational input distribution is seldom known. So-called "random testing" in which test points are chosen from a uniform input distribution is altogether different, because such points could fall in a low-density portion of the

operational distribution, making their selection so unlikely that they should not be counted. Nor is there any way to apply a "safety factor" and use the easier idea of random testing. When most inputs are unlikely (a common case, particularly where data is derived from some real-world sensor like a radar), no safety factor F can insure that in nF inputs there are n that might have been drawn from the operational distribution.

The intuitive reason that unlikely-in-practice test points are not useful is that they may invoke rather different parts of the program code, creating different internal program data states, than in actual practice. Furthermore, thinking this way suggests why the simple theory gives results that seem so at odds with experience: inputs chosen independently may be treated by the program in exactly the same way. To try many such points will be effective only in case one is assured that in use they will actually occur frequently. Another way to say this is: high confidence for high probability of success in operation is unrelated to correctness. A program containing faults may indeed be reliable in practice, because the volume of input required to excite them does not occur.

A theory of random testing that does not rely on the operational input distribution must consider fault distribution over the text of the software in place of test distribution over the input space. Furthermore, to analyze "exercising" methods, the theory must account for the fact that executing faulty code does not necessarily detect the fault. In the simplest theory, faults occur across the source text according to some distribution, and the number of distinct executions required to expose a fault also has a probability distribution. "Distinct executions" means different computation histories in which a given control point appears. The latter assumption is a poor one, because a distribution fails to capture important differences in the way control points are reached, and in the range of computations that include them. Furthermore, many faults are of omission--it is unclear whether "point of omission" is a well defined idea, or whether it makes sense to talk about an omission point being reached in a computation. Nevertheless, a simple theory of this kind might answer questions that resist analysis with logical or algebraic methods.

A program being tested can be instrumented to record the occurrence of each control point in distinct computation histories, yielding a list of the number of times each control point has been encountered during the test. If faults are distributed over N control points $\{c_i\}^N_1$ of a program, and the number of distinct executions to expose a fault is a distribution with mean R, then the probability p_i that an error arise in the test as a result of a fault at control point c_i which has been encountered h_i times can be calculated as the product of the probability of a fault, and the sum of the detection distribution up to h_i. The probability p_E that the test expose at least one error is the sum of p_i over all N control points, and $1 - p_E$ is the probability that no errors appear.

The formula for the probability that no errors appear in a test behaves properly when all of the exercise frequencies are much larger than the mean R required to detect a fault, and when they are much smaller than R or 0. The former case approaches exhaustive testing, and the probability of no errors should be 0; the latter case represents testing with almost no code coverage, and the chance that no errors will appear is 1. In actual testing, it is observed that a test is a mixture of these cases: some statements are well exercised, while others cannot be. Whether it is better to increase already good coverage, or try to improve poor coverage, depends on the particular distributions for fault and detection probabilities.

In testing software for confidence, no errors <u>are</u> observed. Any such test

might be viewed as a sample, and a number of samples might increase confidence that the no-error probability calculated is correct. However, since each sample will yield a different no-error probability, and the independence of samples is doubtful, establishing confidence limits seems a much more difficult problem. Without such limits, however, we cannot hope to explain facts such as software that passes acceptance tests, yet contains residual faults.

Because execution coverage plays an explicit role in this theory, it can be used to analyze testing methods involving coverage metrics--the variants of path testing, mutation testing, etc. For example, consider a test that achieves statement coverage and one that in addition attains branch coverage. It is not unreasonable to conjecture that this improvement is accomplished by adding a nearly optimum set of tests, in which case the changes in the h_i can be estimated, and the methods directly compared.

To increase the probability that errors will show up in a test, it may be necessary to force detailed exploration of narrow subsets of the input space, because only those inputs cause execution histories that reach important code areas. This suggests an input distribution induced by the code: that distribution from which a randomly-selected input is equally likely to cause execution of any program control point. The extent to which this reflected distribution is nonuniform on the input space is a measure of the program's semantic complexity, and of the extent to which input-space random testing will be misleading. The deviation from uniformity can be experimentally determined by dividing the program and the input space into intervals, selecting points at random from the latter, and noting the penetration into the former. This input distribution, unlike the operational one, is connected to the program's *correctness*, in that continued test success on inputs selected from it reduces the likelihood that faults remain.

5. Summary

Testing theory has explanation as its primary goal. Its ideas come from program proving, from practical software tools, and from probability. Only the latter seems promising as a way of analyzing test methods. Results from proving theory substitute noneffective ideas that incorporate tests for the noneffective notion of a correctness proof. When practical approximations to these ideas are considered, the argument that the approximation is useful is only an analogy. Similarly, although modified notions of correctness can be automatically established, the relationship of the established idea to the original is given by a weak plausibility argument. Yet in practice many of the testing methods we use do seem to work very well. Part of this can be explained by the nose-rubbing effect, but part requires analysis that can only be probabilistic in nature. A probabilistic theory must take into account the structure of the code being tested as well as explore the input space [19].

References

1. J. Goodenough and S. Gerhart, Toward a theory of test data selection, Proc Int. Conf. on Reliable Software, Los Angeles, 1975, 493-510.

2. W. Howden, Reliability of the path analysis testing strategy, IEEE Trans. Software Eng. SE-2(1976), 208-215.

3. M. Geller, Test data as an aid in proving program correctness, CACM 21 (May, 1978), 368-375.

4. J. Darringer and J. King, Applications of symbolic execution to program testing, Computer 11 (April, 1978), 51-60.

5. R. DeMillo, R. Lipton, & F. Sayward, Hints on test data selection: help for the practicing programmer, Computer 11 (April, 1978), 34-43.

6. R. Hamlet, Testing programs with the aid of a compiler, IEEE Trans. Software Eng. SE-3 (1977), 279-290.

7. G. Myers, The Art of Software Testing, Wiley, 1979.

8. J. Duran and J. Wiorkowski, Toward models for probabilistic program correctness, Proc. ACM Software Quality & Assurance Workshop, San Diego, 1978.

9. A. R. Meyer and D. M. Ritchie, The complexity of LOOP programs, Proc. 22nd ACM National Conf., 1967, 465-469.

10. R. Hamlet, A patent problem for abstract programming languages: machine-independent computations, Proc. 4th Symp. on Theory of Computing, Denver, 1972, 193-197.

11. M. Minsky, Computation Theory: Finite and Infinite Machines, Prentice-Hall, 1967.

12. P. D. Summers, A methodology for LISP program construction from examples, JACM 24 (Jan., 1977), 161-175.

13. E. Weyuker, Assessing test data adequacy through program inference, TOPLAS, to appear.

14. H. Samet, A canonical form algorithm for proving equivalence of conditional forms, Inf. Proc. Letters 7 (Feb., 1978), 103-106.

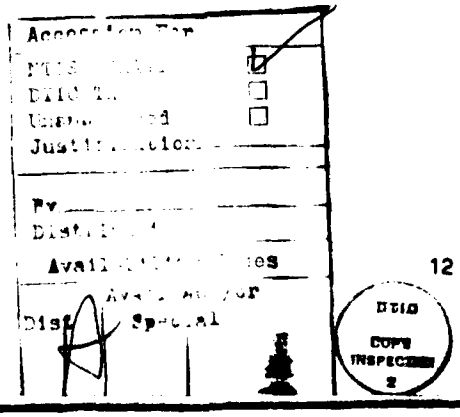
15. D. Richardson and L. Clarke, On the effectiveness of the partition analysis method, IEEE Workshop on Effectiveness of Testing and Proving Methods, Avalon, CA, May, 1982.

16. J. Gannon et al., Data abstraction implementation, specification, and testing, TOPLAS 3 (July, 1981), 211-223.

17. E. Nelson, Estimating software reliability from test data, Microelectonics and Reliability 17 (1978), 67-74.

18. C. Ramamoorthy & F. Bastani, Software reliability—status and perspectives, IEEE Trans. Software Eng. SE-8 (1982), 354-371.

19. M. H. Belz, Statistical Methods for the Process Industries, Macmillan, 1973.



END

DATE
FILMED

5 83

DTIC